International Journal of Information Security
https://doi.org/10.1007/s10207-020-00491-x

REGULAR CONTRIBUTION

Do not let Next-Intent Vulnerability be your next nightmare: type system-based approach to detect it in Android apps

Mohamed A. El-Zawawy[1] · **Eleonora Losiouk**[2] · **Mauro Conti**[2]

Abstract
Android is currently the most widespread operating system (OS) worldwide, but also the most prone to attacks. Despite the challenges faced by Industry and Academia to improve the Android OS security, it still has several vulnerabilities. Among those, the severity of the Next-Intent Vulnerability (NIV) can be immediately grasped. Android apps are made of components, which by default are private and cannot be targeted by other apps on the same phone. However, NIV allows any app to access the private components of a different app, eventually generating a crash or stealing sensitive data. NIV occurs when there is a chain of calls among different components based on the `Intent` messaging model and there is no control over the reliability of the first component triggering the call. NIV was first detected in 2013, but it is still an open issue. In this paper, we present Next-Intent Vulnerability Detector ($\mathcal{NIVD}$), a novel approach to detect NIV in Android apps by relying on type systems. $\mathcal{NIVD}$ applies the inference rules of its type system to the app execution paths containing a sequence of calls to three NIV-related Android APIs. Compared to the state-of-the-art, $\mathcal{NIVD}$ is faster and more efficient, without losing precision in detecting NIV. Finally, through $\mathcal{NIVD}$ *Google Photos* was found to be vulnerable, and we disclosed the finding on the Google official bug report website (issue number 124342801).

Keywords Next-Intent Vulnerabilities · Android · Security · Type systems · Program analysis

1 Introduction

Since its first release, the popularity and the easiness of use of the Android OS have triggered the attention of attackers, that have been focusing more on the Android OS than on other mobile OSs. Even if researchers started immediately addressing the Android OS vulnerabilities proposing solutions to solve them, today their overall number has reached the value of 2000 [1–3]. This is an astonishing result, especially if we consider that each vulnerability is a surface that can be exploited by an attacker to damage the end-user. Moreover, the number of vulnerabilities is continuously growing (125 in 2015 and 611 new ones in 2018 [2]) and new types are emerging. Considering the complexity of the Android internal architecture, that involves several layers from kernel to application, the vulnerabilities can be classified according to the specific layer they address [3]:

1. hardware (e.g., the CVE-2018-9442 regarding the RAM page vulnerability found in hardware components, such as LPDDR2, LPDDR3, or LPDDR4)
2. hardware abstraction (e.g., the WPA2 Wi-Fi security protocol)
3. kernel (e.g., the CVE-2013-6282 regarding the Kernel API `put_user/get_user` vulnerability found in kernel APIs)
4. application (e.g., permission system and inter-process communication)

A recently identified Android security issue, first used to build an attack in [4], is NIV, which allows malicious apps to access to the private components of other apps installed on the same mobile device. NIV originates from the Android communication protocol between two apps, based on the

✉ Mohamed A. El-Zawawy
maelzawawy@cu.edu.eg

Eleonora Losiouk
elosiouk@math.unipd.it

Mauro Conti
conti@math.unipd.it

[1] Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt

[2] Department of Mathematics, University of Padua, Padua, Italy

Published online: 06 March 2020

exchange of `Intent` messages, and from the lack of control on the trustworthiness of the component sending a message. By default, an app can invoke only public components of another app, while NIV enables also the invocation of the private ones. This happens when there is a chain of calls between different Android components and, more specifically, when a public component of an app is supposed to receive an `Intent` from another component and to use it to invoke a private component. This `Intent` can come from any location, either a component of a different app or a component of the same app. Thus, a malicious app can easily exploit this scenario by creating an `Intent` (i.e., the *anchor Intent*), that contains another `Intent` (i.e., the *next Intent*) aimed at invoking the private component. If the target app does not check the `Intent` received, the malicious app can compromise its whole functionality. Among the possible exploits, the malicious app can:

- force the private component to load the URL of a malicious web page. For example, the *PasswordResetActivity* of the *Twitter* app loads the URL contained in its launching `Intent`.
- Fill in the fields of an `Activity` with certain predefined values (e.g., the email address in an email client app).
- Cause the crash of the victim app by sending *Null* data within the *next Intent*.

NIV works on the application-layer, and it can be classified as a bypass authentication vulnerability since the victim app lacks controls over the `Intent` used to invoke a private component. Bypass vulnerabilities represent an increasing threat for Android apps [1–3]: in 2009, there was only one vulnerability of this kind, while in 2018 17 different ones were found. NIV was originally detected in 2013 in apps such as Dropbox and Facebook [4], but it is still an open issue, since little research has been done on similar vulnerabilities [5] and it has been focused on other issues, such as mistakenly defining a private component as public [6,7]. Surprisingly, we found that all Android versions are affected by NIV. This is due to the intrinsic challenge of NIV, which detection requires a data flow tracking system. One possible direction toward the prevention of NIV would be to equip the Android OS with a taint tracking system, which is able to analyze the data flow and detect when an `Intent` sent by an external component is used to launch a private component of another app. Currently, the only available solutions are research contributions (e.g., TaintDroid [8], FlowDroid [7]), which have never been integrated into the Android official framework. Moreover, the detection mechanisms proposed so far analyze Android apps [9] through static analysis techniques, but they do not provide strong defense mechanisms against this threat.

In this paper, we propose a new approach, called $\mathcal{NIVD}$, which statically analyzes an Android app to determine whether it is affected by NIV. In particular, $\mathcal{NIVD}$ inspects the smali code of an Android app and it applies a type system to detect a specific sequence of calls to the NIV-related APIs. According to the Android APIs called along an execution path, $\mathcal{NIVD}$ returns a Boolean value, to express the presence or absence of NIV. We developed a prototype of the designed solution and evaluated it over 100 apps downloaded from the Google Play Store. Among those, we found that nine apps are affected by NIV, including the *Google photos* app, and we reported the issue about *Google photos* to Google (issue number 124342801).[1]

Contributions The contributions of the paper are as follows:

1. We designed and developed $\mathcal{NIVD}$, a novel NIV detection system, which outperforms the state of the art in terms of efficacy (i.e., number of apps found to be affected by NIV) and efficiency (i.e., time required to analyze an app);
2. We identified the building blocks of NIV and we measured their occurrence in the set of 100 Android popular apps;
3. We detected NIV in the *Google Photos* app, which was submitted as a responsible disclosure to Google (issue number 124342801).

Organization The rest of the paper is organized as follows. Section 2 presents the Android `Intent` messaging model and the type system methodology. Section 3 introduces the threat model based on NIV, while the design of the proposed solution is illustrated in Sect. 4. Section 5 presents the implementation details of the $\mathcal{NIVD}$ system, while Sect. 6 shows the achieved results after applying $\mathcal{NIVD}$ on a set of 100 apps. Section 7 presents the comparison to the state-of-the-art, which is deeply discussed in Sect. 8. Finally, Sect. 9 concludes the paper and provides future directions.

2 Background

The purpose of this section is to provide background knowledge about the Android `Intent` messaging model (Sect. 2.1), which is mandatory to understand NIV, and about type systems (Sect. 2.2).

[1] https://bughunter.withgoogle.com/profile/ee300894-5b51-4be6-810f-bcaad8fd73d0.

2.1 Android Intent messaging model

The source code of Android apps is written either in Java or in Kotlin programming language, then translated into Dalvik bytecode and, finally, compressed in a single APK file together with the resources and the `AndroidManifest.xml` file.

Android apps can contain up to four components (i.e., `Activity`, `Service`, `Broadcast Receiver` and `Content Provider`), which are declared in the `AndroidManifest.xml` file and registered by the Android OS at the installation time. By default, Android components are private and can be invoked only by other components belonging to the same app. However, if components are declared as public (i.e., by setting the `exported` property of the component to "true" or by declaring an `Intent Filter` for that component), they can be invoked also by components of other apps installed on the same device. The invocation of a component occurs through the `Intent` messaging model [10] and through specific Android APIs (i.e., `startActivity()`, `startService()` and `sendBroadcast()`). `Intent` objects are messages exchanged between components, which contain either the clear name of the target component (i.e., explicit `Intent`) or the property which the target component should have previously subscribed for (i.e., implicit `Intent`). In addition, `Intent` objects can also carry data, which are then extracted and processed by the target component through the Android APIs `getIntent()` and `getParcelableExtra()`.

2.2 Type system

A type system consists of a set of properties, called types (e.g., `String`, `Integer`), and a set of inference rules used to prevent the execution of operations and instructions on specific types of variables. Each rule is defined by a pre-condition and a post-condition and whenever a program satisfies a pre-condition, it has to satisfy the post-condition to prevent the type system from triggering an error. The application of a type system involves first an assignment of types to variables and, then, a check on the type constraints.

As an illustrative example for introducing type systems, we consider the code shown in Listing 1, where the sum of two variables, an array of `Integer` and an `Integer`, is saved into a third variable of type `Double`.

Listing 1 Sample code for applying a type system

```
1  void Type_program()
2  {
3  int x[10],y;
4  double z;
5  z:=x[3]+y;
6  }
```

Table 1 Inference rules of the type system example

$$\frac{i \text{ is an integer}}{\Gamma \models i : int} \quad (1)$$

$$\frac{x : T \in \Gamma}{\Gamma \models x : T} \quad (2)$$

$$\frac{\Gamma \models e_1 : int \quad \Gamma \models e_2 : int}{\Gamma \models e_1 + e_2 : int} \quad (3)$$

$$\frac{\Gamma \models e_1 : T[] \quad \Gamma \models e_2 : int}{\Gamma \models e_1[e_2] : T} \quad (4)$$

$$\frac{\Gamma \models e_1 : T_1 \quad \Gamma \models e_2 : T_2 \quad T_2 \leq T_1}{\Gamma \models e_1 := e_2 : T_2} \quad (5)$$

In this example, the purpose of our type system is to check whether the types of the three variables involved in the sum operation are consistent. Thus, we first define the set of types $T = \{int, double\}$ and, then, the inference rules shown in Table 1. For each rule, the pre-condition is above the line and the post-condition is below the line and the symbol Γ denotes the typing environment function, which assigns a set of types to a set of variables. The meaning of the rules is as follows: Rule 1 claims that, if i is an integer constant, then its type is `int`; Rule 2 says that the type of x is T if x is created as a T object; Rule 3 claims that the sum of two `Integer` variables is still an `Integer` variable; Rule 4 says that, if we have an array of elements of type T and an `Integer` variable used as an index of the array, the element of the array will be of type T; Rule 5 claims that the assignment $e_1 := e_2$ is of *type* T_2 if e_1 has type T_1, e_2 has type T_2, and $T_2 \leq T_1$. Thus, the pair (T, R) defines our type system and it can be used to type-check the program example in Listing 1.

Table 2 shows how the type system is applied to the sample code. The type checking process starts from Line 3 of the program, where the variables x and y are declared. We note that at the start of this process the typing environment Γ is empty. At this point, the type system reflects the declarations in the typing environment Γ by assigning a type to each variable. Thus, after Line 3, Γ becomes $\{x : int[], y : int\}$, where x is assigned the type `Array` and y the type `Integer`. Similarly, after Line 4, Γ becomes $\{x : int[], y : int, z : double\}$. Then, on Line 5 the program defines an assignment which is treated by Rule 5. This rule has three terms in its pre-condition. The first term is z, whose type has been already determined. The second term is $x[3]$, which triggers Rule 4, that again invokes Rule 2 and Rule 1. Finally, when all types of the three terms are identified, the pre-condition of Rule 5 can be evaluated.

The above-mentioned type system prevents type errors such as the following one. If we suppose that z is declared as a `String`, then the inequality $int \leq string$ in the pre-condition of Rule 5 is not satisfied and the type checking process aborts.

Table 2 Application of the type system on the sample code

$$\dfrac{\dfrac{c\ z : double \in S}{\Gamma \models z : double}\,(by\ 2) \qquad \dfrac{\dfrac{\dfrac{x : int[] \in S}{\Gamma \models x : int[]}\,(by\ 2) \qquad \dfrac{3 \text{ is an integer}}{\Gamma \models 3 : int}\,(by\ 1)}{\Gamma \models x[3] : int}\,(by\ 4) \qquad \dfrac{y : int \in S}{\Gamma \models y : int}\,(by\ 2)}{\Gamma \models x[3] + y : int}\,(by\ 3) \qquad int \leq double}{\Gamma = \{x : int[], y : int, z : double\} \models z := x[3] + y : int}\,(by\ 5)$$

3 Threat model

Our threat model involves a malicious Android app, that aims at invoking the private component of another app installed on the same mobile device. The malicious app does not require any permission, since it exploits the NIV of the victim app to complete its attack.

An app is considered NIV vulnerable if the following scenario occurs. The app has two different components, a public and a private one, respectively. When the public component is invoked and executed, it extracts the additional data contained in its launching `Intent` and invokes the private component. In particular, the additional data contains another `Intent`, which is then forwarded to the private component. As before, the private component can extract the additional data coming within its launching `Intent`.

To better illustrate how an attacker can exploit NIV to damage a victim app, we will refer to the example shown in Fig. 1. In this attack scenario, we suppose to have two separate apps installed on the same mobile device: the victim app, which declares at least a public and a private component, and the malicious app, which aims at communicating with the private component of the victim app. To achieve its aim, the malicious app sends the *Intent M*, which targets *pu*, the public component of the victim app. In addition, the *Intent M* is equipped with another `Intent` object (*Intent M′*), which targets *pr*, the private component of the victim app. Thus, when *pu* receives *Intent M*, it extracts *Intent M′* and forwards it to *pr*. At this point, *pr* might have been designed to extract and process the additional data from its launching `Intent` *Intent M′*. This way, the attacker is able to invoke and send data to a private component of the victim app.

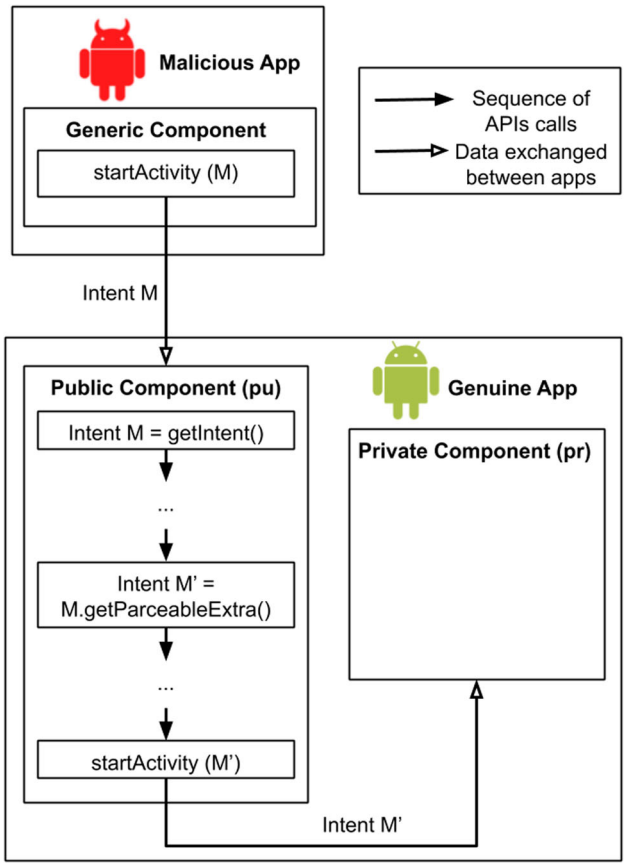

Fig. 1 Example of NIV malicious exploitation

4 $\mathcal{N}$IV$\mathcal{D}$: design

In this section, we describe in details the $\mathcal{N}$IV$\mathcal{D}$ solution, starting from a complete overview (Sect. 4.1), moving to its two Algorithms (Sect. 4.2) and concluding with the type system (Sects. 4.3, 4.4), together with an illustrative example (Sect. 4.5).

The current design of $\mathcal{N}$IV$\mathcal{D}$ focuses on `Activities` only, since previous works [5] highlighted that the 85% of Android NIV vulnerable components are `Activities`.

4.1 $\mathcal{N}$IV$\mathcal{D}$ overview

Figure 2 shows the complete workflow of $\mathcal{N}$IV$\mathcal{D}$, which starts with the acquisition of an APK file and finishes with a report containing the details of the NIV detection. As a starting point, $\mathcal{N}$IV$\mathcal{D}$ decompiles the APK file through *Androguard* [11] (Step 1 in Fig. 2). Then, $\mathcal{N}$IV$\mathcal{D}$ searches for the public `Activities` of the app since those are likely to be the starting point of a path affected by NIV (Step 2). Once

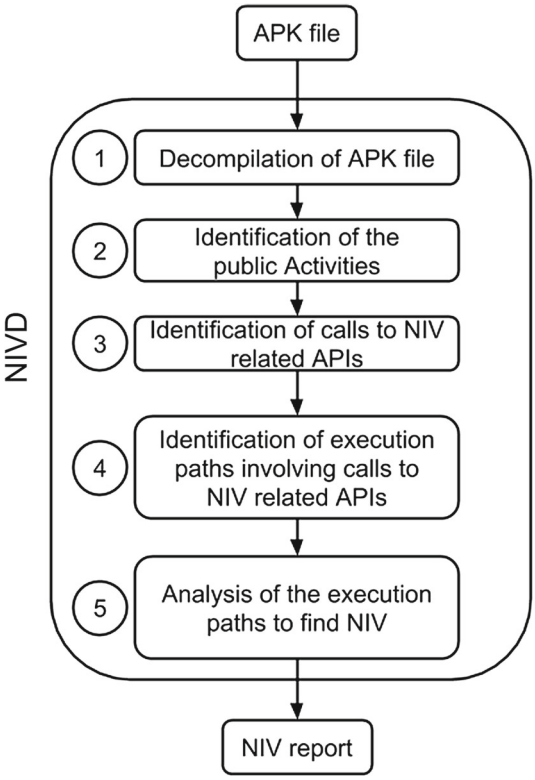

Fig. 2 Overview of the $\mathcal{N}$I$\mathcal{V}$D system

the `Activities` are found, $\mathcal{N}$I$\mathcal{V}$D checks whether these components contain calls to the Android NIV-related APIs (Step 3). If this is the case, $\mathcal{N}$I$\mathcal{V}$D identifies the complete execution paths containing the calls to the above-mentioned APIs (Step 4) and, finally, analyzes each of them to detect NIV (Step 5).

4.2 $\mathcal{N}$I$\mathcal{V}$D algorithms

$\mathcal{N}$I$\mathcal{V}$D consists of two algorithms: a first one, that identifies all execution paths possibly affected by NIV, and a second one, that applies a type system on an execution path to confirm whether NIV is present.

Algorithm 1 receives an APK file, finds all possible NIV vulnerable paths, checks the vulnerability of each path by invoking Algorithm 2 and, finally, returns either the *Secure* value, if no NIV has been detected, or the *Not Secure* value, with the vulnerable paths. NIV is detected whenever the NIV-related Android APIs are called in the following order: `getIntent()`, `getParcelableExtra()` and `startActivity()`. More specifically, this sequence of calls occurs when an `Activity` retrieves its launching `Intent` (i.e., `getIntent()`), extracts the nested `Intent` (i.e., `getParcelableExtra()`) and, then, starts a new `Activity` with the nested `Intent` (i.e., `startActivity()`). Between the invocation of the NIV-related APIs, the app might also invoke other Android APIs, which are, however, ignored by $\mathcal{N}$I$\mathcal{V}$D . The only assumption of $\mathcal{N}$I$\mathcal{V}$D is that the call to `getIntent()` should be done by an `Activity` component.

Algorithm 1 starts first with the decompilation of the APK file (Line 1), then with the search of all public `Activities` of the app (Line 2) and it moves on with the identification of all invocations of the `getIntent()` API (Line 3). If there is no invocation to `getIntent()` (Line 4), $\mathcal{N}$I$\mathcal{V}$D terminates, returning the *Secure* value (Line 5). Otherwise, Algorithm 1 analyzes again the application, searching for all invocations to `getParcelableExtra()` (Line 6). If no invocations are found, $\mathcal{N}$I$\mathcal{V}$D terminates, returning the *Secure* value (Line 8). Once again, the algorithm inspects the app to search for all invocations of `startActivity()` (Line 9). If no invocations are found, $\mathcal{N}$I$\mathcal{V}$D terminates, returning the *Secure* value (Line 11). Otherwise, Algorithm 1 saves in φ all paths having the invocations of the NIV-related APIs in the expected order (Line 12). If the set φ is found empty (Line 13), the app is considered *Secure* (Line 14). Otherwise, $\mathcal{N}$I$\mathcal{V}$D calls Algorithm 2, which analyzes each path in φ to verify if it is NIV vulnerable (Line 16-18). Each vulnerable path is saved into $\mathcal{P}$, which is then returned at the end of the execution (Line 19 and 20).

Algorithm 1

Input: The APK file of the application a.
Output: "Secure" if $\mathcal{A}$ has no NIV; ("Not Secure", $\mathcal{P}$) if $\mathcal{A}$ has NIV. $\mathcal{P}$ is the set of paths that have NIV.
Steps:

1: $\mathcal{A} \leftarrow$ decompilation of a using *Androguard*.
2: $\delta \leftarrow$ public activities of $\mathcal{A}$.
3: $\chi_1 \leftarrow$ locations of `getIntent()` instruction calls.
4: **if** $\chi_1 = \emptyset$ **then**
5: **return Secure**;
6: $\chi_2 \leftarrow$ locations of `getParcelableExtra()` instruction calls in $\mathcal{A}$.
7: **if** $\chi_2 = \emptyset$ **then**
8: **return Secure**;
9: $\chi_3 \leftarrow$ locations of `startActivity()` instruction calls in $\mathcal{A}$.
10: **if** $\chi_3 = \emptyset$ **then**
11: **return Secure**;
12: $\varphi \leftarrow$ the set of paths that start at $l_1 \in \chi_1$, go through $l_2 \in \chi_2$, and then go through $l_3 \in \chi_3$ in $\mathcal{A}$.
13: **if** $\varphi = \emptyset$ **then**
14: **return Secure**;
15: $\mathcal{P} \leftarrow \emptyset$;
16: **for each** $p \in \varphi$ **do**
17: **if** *Algorithm 2(p)* **then**
18: $\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}$;
19: **if** $\mathcal{P} \neq \emptyset$ **then**
20: **return** (**Not Secure**, $\mathcal{P}$);
21: **else**
22: **return Secure**;

Algorithm 2 receives an execution path and relies on a type system to verify whether the path is affected by NIV. At first, Algorithm 2 initializes the required set of variables: the typing environment Γ_0 (Line 1), the variable s, which saves whether the component is public or private (Line 2), the instruction counter c (Line 3), the variable *prev_in*, which saves a pointer to the previous instruction (Line 4). Then, the algorithm iterates over the set of instructions contained in p (Line 5), and for each one, it applies the appropriate rule of the type system (Line 6). If NIV has been detected (Line 7), $\mathcal{N}\mathcal{I}\mathcal{V}\mathcal{D}$ immediately terminates returning a *True* value (Line 8). Otherwise, the algorithm continues with the evaluation of the next instruction (Line 9–10).

Algorithm 2

Input: An execution path, p, of an application.
Output: "True" if p has NIV; "False" otherwise.
Steps:
1: $\Gamma_0 \leftarrow \emptyset$
2: $s \leftarrow$ `getScopeType(p);`
3: $c \leftarrow 0$;
4: *prev_in* $\leftarrow$ *none*;
5: **for each** $in \in p$ **do**
6: find (Γ_{c+1}, t_c) such that $(s, prev\_in) \models in : \Gamma_c \rightarrow (\Gamma_{c+1}, t_c)$
7: **if** $t_c ==$ *NIV* **then**
8: **return** *True*
9: $c \leftarrow c + 1$;
10: *prev_in* $\leftarrow$ *in*;
11: **return** *False*

4.3 Types of the $\mathcal{N}\mathcal{I}\mathcal{V}\mathcal{D}$ type system

To describe the inference rules of the $\mathcal{N}\mathcal{I}\mathcal{V}\mathcal{D}$ type system, we first illustrate the syntax shown in Table 3. The Dalvik instructions are executed in the Dalvik Virtual Machine (DVM), which is a register based machine. Thus, registers are the input variables for Dalvik instructions and in our type system they are indicated with the v_i $(i \in \mathbb{N})$ variable. Each register may assume different values and, among those, our type system considers the following ones: *null* | *true* | *false* | v | *Intent*. The variables c_{get}, c_{put} and c_{type} are used to represent a set of Dalvik instructions aimed at reading or writing values. In particular, the instructions are represented through the concatenations $c_{get}c_{type}$ and $c_{put}c_{type}$, where c_{get} refers to the reading instructions (i.e., *aget-*, *iget-*, *sget-*), c_{put} to the writing instructions (i.e., *aput-*, *iput-*, *sput-*) and c_{type} to the common data types (i.e., *wide*, *object*, *boolean*, *byte*, *char*, *short*). The variable in refers to the current instruction under analysis, while s saves whether in belongs either to a public or a private component. Each variable in our type system has its own type $t \in T$. In addition to the standard types (e.g., `String`, `Integer`) we also define two custom types: the register type $t_r \in T_r$ and the instruction type $t_i \in T_i$. The first one can contain either the triple (p_1, p_2, t_r) or the *Null* value. In the first case, it means that the register holds an `Intent` whose properties are p_1, p_2, and t_r; otherwise, the register does not hold an `Intent`. The symbol p_1 specifies whether the `Intent` was created in a public or private component, while p_2 specifies whether the `Intent` is an *anchor* or a *next* one. Finally, the symbol t_r is the type of the extra data carried by the `Intent`. The instruction type is either *NIV*, if NIV has been detected in the execution path up to the current instruction, or *Secure* otherwise. Finally, Γ is the typing environment, that matches a register v to the type t_r.

To design our type system, we first have to consider the sequence of Android APIs reflecting NIV (i.e., `getIntent()`, `getParcelableExtra()` and

Table 3 Types and syntax of the $\mathcal{N}\mathcal{I}\mathcal{V}\mathcal{D}$ type system

Variable		Variable value	Variable description
V	::=	v_i $(i \in \mathbb{N})$	Register
c_{get}	::=	*aget-* | *iget-* | *sget-*	
c_{put}	::=	*aput-* | *iput-* | *sput-*	
c_{type}	::=	*wide* | *object* | *boolean* | *byte* | *char* | *short*	
in	::=	... | *new-instance*(v_{dest}, v_{type}) | ...	Instruction under analysis
s	::=	Public | private	Whether the instruction belongs to public or private component
$t \in T$	::=	t_r | t_i	Type
$t_r \in T_r$	::=	(p_1, p_2, t_r) | null	Register type
$t_i \in T_i$	::=	NIV | Secure	Instruction type
$p_1 \in P_1$	::=	Public | private	Intent property
$p_2 \in P_2$	::=	Anchor | next	Intent property
$\Gamma \in \mathcal{E}$	::=	$v_i \rightarrowtail t_r$	Typing environment

`startActivity()`). These APIs are expressed in the Android programming language, while our type system works on Dalvik instructions. Thus, we analyzed the complete set of Dalvik instructions [12] to design a type system able to detect NIV. In particular, we considered the following instructions:

- $new-instance(v_{dest}, v_{type})$: creates an object of type v_{type} and saves its address in v_{dest}
- $const*(v_{dest}, v_{source})$: assigns the constant value saved in v_{source} to v_{dest}
- $move*(v_{dest}, v_{source})$: moves the value specified in v_{source} into v_{dest}
- $c_{get}c_{type}(v_{dest}, v_{source1}, v_{source2})$: copies an object determined by $v_{source1}$ and $v_{source2}$ into v_{dest}
- $c_{put}c_{type}(v_{source}, v_{dest1}, v_{dest2})$: copies a value from v_{source} into the field specified in v_{dest1} and v_{dest2}
- $invoke*(v_{object}, \ldots, v_{API})$: denotes a set of instructions used for invoking the API specified in v_{API} on the object saved in v_{object}
- $move-result*(v_{dest})$: copies the result of the most recent API invoked into v_{dest}

4.4 Rules of the $\mathcal{NIVD}$ type system

Table 4 presents the 13 inference rules we designed to detect NIV. Each rule has a pre-condition and a post-condition and the significant output is the final instruction type identified by applying the following pattern:

$$(s, in_p) \models in : \Gamma_{old} \rightarrow (\Gamma_{new}, t_i)$$

where s specifies whether the current instruction belongs to a public or private component, in_p is the last instruction executed, in is the current instruction under analysis, Γ_{old} and Γ_{new} are the typing environments identified before and after the execution of in, t_i is the instruction type that specifies whether in is *Secure* or not.

Rule 1 This rule detects the creation of a new object of type v_{type} (i.e., `Intent`), which is saved into v_{dest}. The register type of v_{dest} is (s, *anchor, null*), which means that it inherits its p_1 property from s, it is an *anchor Intent* and it does not hold a *next Intent*. Thus, the final instruction type is *Secure*.

Rule 2 This rule handles the assignment of a constant value to v_{dest}. After the assignment, the register type of v_{dest} is *Null*, which means that the final instruction type is *Secure*.

Rule 3 This rule manages the transfer of a value from v_{source} to v_{dest}. The destination register inherits its register type from the source register. Thus, the final instruction type is *Secure*.

Rule 4 This rule aims at detecting all APIs used for retrieving a value, that is not an object. After the retrieval of the value from $v_{source1}$ and $v_{source2}$ and its saving into v_{dest}, the register type of the destination register is *Null*. Thus, the final instruction type is *Secure*.

Rule 5 This rule detects all methods enabling the retrieval of an object from $v_{source1}$ and $v_{source2}$, which is then saved into v_{dest}. After the retrieval, v_{dest} inherits its register type from $v_{source1}$, which means that the final instruction type is *Secure*.

Rule 6 The rule detects the set of instructions that write a value, which is not an object, from v_{source} into v_{dest1} and v_{dest2}. As a result, the new register type of v_{dest1} is *Null* and the final instruction type is *Secure*.

Rule 7 The rule handles the set of instructions that write an object, retrieved from v_{source}, into v_{dest1} and v_{dest2}. In this case, v_{dest1} inherits its new register type from v_{source} and the final instruction type is *Secure*.

Rule 8 The rule detects the invocation of the `startActivity` method and, more specifically, when the invocation involves a public, *next Intent* as an argument. Since this is the actual point where NIV occurs, the final instruction type is *NIV*.

Rule 9 This rule detects the invocation of the `putExtra` API which enables to save an object (i.e., v_{object}) as extra data of another register (i.e., v_{dest}). The rule considers the specific case where both v_{object} and v_{dest} are `Intent`. The invocation of `putExtra` is reflected in the new register type of the third element of v_{dest}, which becomes the same as the one of v_{object}. Thus, the final instruction type is *Secure*.

Rule 10 The rule manages the transfer of a value, retrieved after invoking an API on $v_{prevobject}$, into v_{dest}. In this case, the detected API is the `getParcelableExtra` method and the $v_{prevobject}$ is an *anchor Intent*. According to the semantics of the API, v_{dest} inherits its new register type from the third element of $v_{prevobject}$. As a result, the final instruction type is *Secure*.

Rule 11 The rule manages the transfer of a value, retrieved after invoking an API on $v_{prevobject}$, into v_{dest}. In this case, the detected API is the `getIntent` method and the new register type of v_{dest} is (s, *anchor, (s, next, null)*), which means that it is an *anchor* `Intent` and, consequently, the final instruction type is *Secure*.

Rule 12 This rule detects all set of APIs, that do not return an `Intent`. The new register type of v_{dest} is *Null* and the final instruction type is *Secure*.

Rule 13 This rule considers all instructions, that are not detected by the previous rules and that do not affect the typing environment. Thus, when any other instruction is detected, the final instruction type is *Secure*.

Table 4 Inference rules of the $\mathcal{N}\mathcal{I}\mathcal{V}\mathcal{D}$ type system

$$\frac{v_{type}=Intent \qquad \Gamma_{new}=\Gamma[v_{dest}\mapsto(s,anchor,\ null)]}{(s,in_p)\models new-instance(v_{dest},v_{type}):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{1}$$

$$\frac{\Gamma_{new}=\Gamma[v_{dest}\mapsto Null]}{(s,in_p)\models const*(v_{dest},\ldots):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{2}$$

$$\frac{\Gamma_{new}=\Gamma[v_{dest}\mapsto\Gamma_{old}(v_{source})]}{(s,in_p)\models move*(v_{dest},v_{source}):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{3}$$

$$\frac{\begin{array}{l}c_{get}\in\{aget\text{-},\ iget\text{-},\ sget\text{-}\}\\ c_{type}\in\{wide,\ boolean,\ byte,\ char,\ short\}\\ \Gamma_{new}=\Gamma[v_{dest}\mapsto Null]\end{array}}{(s,in_p)\models c_{get}c_{type}(v_{dest},v_{source1},v_{source2}):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{4}$$

$$\frac{\begin{array}{l}c_{get}\in\{aget\text{-},\ iget\text{-},\ sget\text{-}\}\\ c_{type}\notin\{wide,\ boolean,\ byte,\ char,\ short\}\\ \Gamma_{new}=\Gamma[v_{dest}\mapsto\Gamma(v_{source1})]\end{array}}{(s,in_p)\models c_{get}c_{type}(v_{dest},v_{source1},v_{source2}):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{5}$$

$$\frac{\begin{array}{l}c_{put}\in\{aput\text{-},\ iput\text{-},\ sput\text{-}\}\\ c_{type}\in\{wide,\ boolean,\ byte,\ char,\ short\}\\ \Gamma_{new}=\Gamma[v_{dest1}\mapsto Null]\end{array}}{(s,in_p)\models c_{put}c_{type}(v_{source},v_{dest1},v_{dest2}):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{6}$$

$$\frac{\begin{array}{l}c_{put}\in\{aput\text{-},\ iput\text{-},\ sput\text{-}\}\\ c_{type}\notin\{wide,\ boolean,\ byte,\ char,\ short\}\\ \Gamma_{new}=\Gamma[v_{dest1}\mapsto\Gamma(v_{source})]\end{array}}{(s,in_p)\models c_{put}c_{type}(v_{source},v_{dest1},v_{dest2}):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{7}$$

$$\frac{v_{API}=startActivity \qquad \Gamma_{new}(v_{object})=(public,\ next,\_)}{(s,in_p)\models invoke*(v_{object},\ldots,v_{API}):\Gamma_{old}\to(\Gamma_{new},NIV)} \tag{8}$$

$$\frac{\begin{array}{l}v_{API}=putExtra \qquad \Gamma_{old}(v_{dest})\ \&\ \Gamma_{old}(v_{object})are\ not\ Null\\ \Gamma_{new}=\Gamma[v_{dest}\mapsto(\Gamma_{old}(v_{dest})[0],\Gamma_{old}(v_{dest})[1],\Gamma_{old}(v_{object}))]\end{array}}{(s,in_p)\models invoke*(v_{dest},\ldots,v_{API}):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{9}$$

$$\frac{\begin{array}{l}in_p=invoke(v_{prevobject},\ldots,getParcelableExtra(\ldots))\\ \Gamma_{old}(v_{prevobject})is\ an\ anchor\ Intent\\ \Gamma_{new}=\Gamma[v_{dest}\mapsto\Gamma_{old}(v_{prevobject})[2]]\end{array}}{(s,in_p)\models move-result*(v_{dest},\ldots):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{10}$$

$$\frac{\begin{array}{l}in_p=invoke(v_{prevobject},\ldots,getIntent())\\ \Gamma_{new}=\Gamma[v_{dest}\mapsto(s,\ anchor,\ (s,\ next,\ null))]\end{array}}{(s,in_p)\models move-result*(v_{dest},\ldots):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{11}$$

$$\frac{\begin{array}{l}in_p=invoke(v_{prevobject},\ldots,API)\\ API\ does\ not\ return\ an\ Intent\\ \Gamma_{new}=\Gamma[v_{dest}\mapsto Null]\end{array}}{(s,in_p)\models move-result*(v_{dest},\ldots):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{12}$$

$$\frac{in=any\ instruction\ not\ treated\ by\ previous\ rules}{(s,in_p)\models in(\ldots):\Gamma_{old}\to(\Gamma_{new},Secure)} \tag{13}$$

4.5 Illustrative example for the $\mathcal{N}\mathcal{I}\mathcal{V}\mathcal{D}$ type system

Listing 2 shows an illustrative example to describe how the $\mathcal{N}\mathcal{I}\mathcal{V}\mathcal{D}$ type system works in detecting NIV in a vulnerable app.

The listing contains Dalvik instructions of an Android app under analysis. In this case, a public `Activity` invokes the `getIntent()` method to retrieve its launching `Intent` (Line 4), which is then saved into v_{d1} (Line 5). Then, in v_{d2} it defines the constant value corresponding to the key used for the extra data in the `Intent` just retrieved (Line 6). The extra data of v_{d1}, available under the v_{d2} key, are obtained through the call to `getParcelableExtra()` (Line 8) and the result is saved into v_{d3} (Line 9). The type of the object just retrieved is checked to see whether it is an `Intent` (Line 10, Line 11) and, if this is the case, the `startActivity` method is invoked on the current `Activity` (i.e., v_{object}) with the `Intent` saved in v_{d3} as an argument.

Listing 3 shows how the inference rules are applied on the set of Dalvik instructions. Starting with an empty typing environment, the invocation of the `getIntent()` method is detected by Rule 13 without any effect on the typing environment (Line 5). The next instruction finds a match with Rule 11 and the new instruction type for v_{d1} is *(public, anchor, (public, next, null))* (Line 7). Even though we are not sure that the extracted `Intent` is an *anchor Intent*, we consider the worst case by applying Rule 11. Moving on, both the assignment of a constant value to v_{d2} and the invocation of the `getParcelableExtra()` method are detected by Rule 13 without any consequence on the typing environment (Line 9 and Line 11). Rule 10 handles the saving of the extra data just extracted into v_{d3} (Line 13). After this instruction, the register type for v_{d3} is *(public, next, null)*, since it is a *next Intent*, while for v_{d2} is *Null*, since it holds a constant value. Rule 13 detects again the next two instructions (Line 15

and Line 17), until the invocation to the `startActivity` method is reached and the Rule 8 handles it (Line 19). Since the method is invoked from a public `Activity`, it receives a *next* `Intent` as an argument, and it might invoke a private component of the app, the new instruction type is classified as *NIV*.

5 Implementation

To develop $\mathcal{N}$I$\mathcal{V}$D , we used Python v3.6.5 as a programming language and here we provide the details of our implementation.

Decompilation of APK file The implementation of $\mathcal{N}$I$\mathcal{V}$D strongly relies on the *Androguard* Python library [11], which enables to decompile the APK file of an app and to extract specific information. In particular, for decompiling an APK we use the `AnalyzeAPK` method, which returns three objects:

- An APK object, which contains all information about the APK, such as the `AndroidManifest.xml` file, permissions, and package name.
- A *DalvikVMFormat* object, which contains classes and methods of the app under analysis.
- An analysis object, which allows to perform further analysis on the app.

Identification of the public Activities To identify all public `Activities` of an app, we first extract all `Activity` components contained in the `AndroidManifest.xml` file. To achieve this aim, we search for the xml elements having the keyword "activity" under the `TagName` by using the `getElementsByTagName()` API. Then, to evaluate whether each identified `Activity` is public, we consider the following conditions:

- The `exported` attribute of the xml element is set to true (through the `getAttributeNS()` API).
- The xml element declares an `Intent Filter`, which is detected through the `getElementsByTagName()` API.

Finally, the classes associated to the public `Activities` are extracted from the *DalvikVMFormat* object.

Identification of calls to NIV-related APIs The calls toward the `getIntent()`, `getParcelableExtra()`, and `startActivity()` APIs are detected by following the same approach, besides the location of the `getIntent()` call, which has to be only in public `Activities`. The Android API detection starts with the `get_methods()` `Androguard` API, which returns all methods of a class. Then, the block of instructions defining each method is obtained through the `basic_blocks.gets()` `Androguard` API and the single instruction of the above-mentioned block through the `get_instructions()` `Androguard` API. Finally, for each instruction we evaluate its set of operands, obtained through the `get_operands()` API, with a specific attention to the last operand, since this is the actual call to an Android API. Thus, if the last operand points to `getIntent()`, it means that the instruction calls the specified API.

For each block of instructions including a call to `getIntent()`, we identify the children blocks (i.e., successor blocks in the possible execution paths) through the `childs` attribute of the block. Then, we compare each child with the previously identified blocks, which contain a call to `getParcelableExtra()` and `startActivity()`. While moving from a parent to a child block, we encounter the overall number of execution paths, until one of the following conditions is verified:

- The child has been already treated.
- The child has no children.
- The number of analyzed execution paths is 65,536, which is a convenient threshold according to previous work [5].

Analysis of the execution paths to find NIV Once the execution paths are identified, we apply the $\mathcal{N}$I$\mathcal{V}$D type system to each one of them. To achieve this aim, we have first to define the context of a block (i.e., whether it belongs to a public or private component) by using the `get_method()` and `get_class_name()` APIs. The typing environments are saved into lists and the initial one is *Null*. Through the `get_operands()`, we detect every time the invoked API. The result of applying a rule of the type system is reflected into the update of the typing environment.

6 $\mathcal{N}$I$\mathcal{V}$D results

The purpose of this section is to illustrate the results obtained after running $\mathcal{N}$I$\mathcal{V}$D on a set of applications downloaded from the Google Play Store (see "Appendix A" for the complete list of apps). In particular, our aim is to determine how many apps, out of the downloaded 100 ones, are affected by NIV and what is the distribution of the NIV building blocks in those apps. The complete set of analyses of the apps is provided in "Appendix B" and raw data are available online.[2]

[2] https://scholar.cu.edu.eg/sites/default/files/maelzawawy/files/NIVDResults.rar.

Listing 2 Smali code of an NIV vulnerable app.

```
1  void My_Public_Activity()
2  {
3  ⋮
4  invoke–virtual(getIntent())
5  move–result–object(v_d1)
6  const–string(v_d2,``launch_on_find_intent)
7  ⋮
8  invoke–virtual(v_d1,v_d2,getParcelableExtra())
9  move–result–object(v_d3)
10 check–cast(v_d3,Landroid/content/Intent;)
11 if–eqz(v_d3,...)
12 ⋮
13 invoke–virtual(v_object,v_d3,startActivity())
14 ⋮
15 }
```

Listing 3 $\mathcal{N}\mathrm{I}\mathcal{V}\mathrm{D}$ type system applied on the smali code of an NIV vulnerable app.

```
1  void My_Public_Activity()
2  {
3  Γ_0 = ∅
4  invoke–virtual(getIntent())
5  (Γ_1 = {}, Secure), by Rule 13
6  move–result–object(v_d1)
7  (Γ_2 = {v_d1 ← (public, anchor, (public, next, null))}, Secure), by Rule 11
8  const–string(v_d2, ``launch_on_find_intent)
9  (Γ_3 = {v_d1 ← (public, anchor, (public, next, null))}, Secure), by Rule 13
10 invoke–virtual(v_d1,v_d2,getParcelableExtra())
11 (Γ_4 = {v_d1 ← (public, anchor, (public, next, null)), v_d2 ← null}, Secure), by Rule 13
12 move–result–object(v_d3)
13 (Γ_5 = {v_d3 ← (public, next, null), v_d2 ← null}, Secure), by Rule 10
14 check–cast(v_d3,Landroid/content/Intent;)
15 (Γ_6 = {v_d3 ← (public, next, null), v_d2 ← null}, Secure), by Rule 13
16 if–eqz(v_d3,...)
17 (Γ_7 = {v_d3 ← (public, next, null), v_d2 ← null}, Secure), by Rule 13
18 invoke–virtual(v_object,v_d3,startActivity())
19 (Γ_8 = {v_d3 ← (public, next, null), v_d2 ← null}, NIV), by Rule 8
20 }
```

We ran the experiments on a Dell (Vostro) device with a processor Intel(R) Core(TM) i7-3612 QM, CPU @ 2.10 GHz, 8.00 GB of RAM, and Windows 10 (64-bits).

The aim of $\mathcal{N}\mathrm{I}\mathcal{V}\mathrm{D}$ is to detect the existence of NIV in Android apps. Thus, we chose 100 apps, among the most popular ones on the Google Play Store, and we ran $\mathcal{N}\mathrm{I}\mathcal{V}\mathrm{D}$ against them. We found nine vulnerable apps, which are shown in Table 5, together with: the number of execution paths containing NIV (i.e., $\mathcal{NIV}_n$), the names of the activities ($\mathcal{NIV}_a$) and of the methods ($\mathcal{NIV}_m$) from which these vulnerable paths start. Among the vulnerable apps, *Google Photos* was found to have a high number of $\mathcal{NIV}_n$: this is due to the branching instructions (i.e., `if` statements) contained in the vulnerable execution paths. As a matter of fact, if a vulnerable execution path is split into two separate branches due to an `if` statement, the final counter of the number of $\mathcal{NIV}_n$ will be two.

In addition to the previous analysis, we also measured the distribution of the NIV building blocks in the same set of apps. In particular, the NIV building blocks involve the sequence of calls to the expected APIs (i.e., `getIntent()`, `getParcelableExtra()` and `startActivity()`) plus the data flow of the *Intent* object, which is used to invoke a private component and is sent by a different app installed on the same smartphone. To measure the NIV building blocks, we defined four different degrees:

Table 5 NIV vulnerable apps found over the set of 100 apps

App (version)	NIV_n	NIV_a	NIV_m
BeeTalk (v3.0.8)	2	BTSplashActivity	Obfuscated method
DB Navigator (v18.04.p04.01)	1	WebAccessActivity	Obfuscated method
Evernote (v6.1)	23	LandingActivity	Obfuscated method
		LandingActivity	onResume
Evernote (v7.0.1)	23	LandingActivity	Obfuscated method
		LandingActivity	onResume
MeetMe (v12.10.1.1265)	6	LaunchActivity	onCreate
Twitter (v7.23.0)	2	LoginActivity	onActivityResult
		VerifyLoginActivity$b	Obfuscated method
Viber Messenger (v8.9.0.2)	2	WesternUnionWelcomeActivity$e	onClick
Viber Messenger (v7.9.2.10)	2	WesternUnionWelcomeActivity$e	onClick
Google Photos (v3.9.0.175053409)	11,662	HostPhotoPagerActivity	Obfuscated method
ES File Explorer(v4.1.9.9.3)	2	NewSplashActivity	Obfuscated method
		NewSplashActivity	Obfuscated method
Google Play Games (v5.14.7825)	52	ClientUiProxyActivity	onCreate

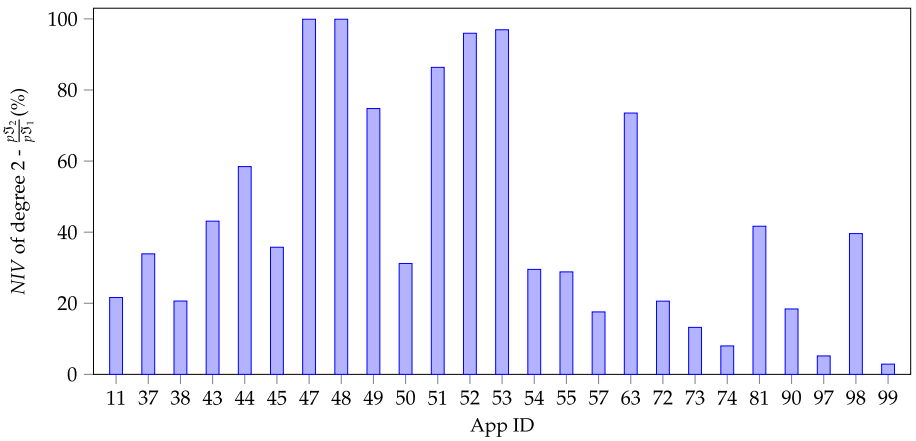

Fig. 3 Apps found to have an *NIV* of degree 2 over the set of 100 apps

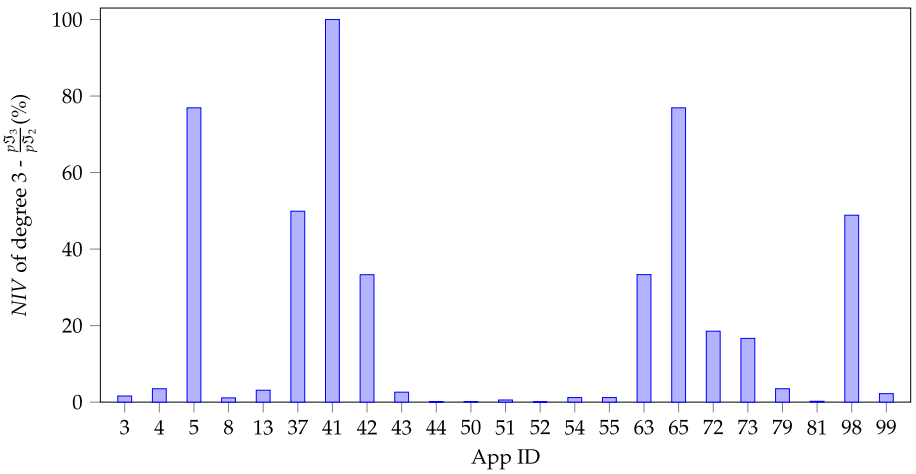

Fig. 4 Apps found to have an *NIV* of degree 3 over the set of 100 apps

– *NIV* of degree 1: if an app contains at least one path calling the `getIntent()` API (i.e., $p\Im_1$ is greater than zero), it is classified to have an *NIV* of degree 1 (e.g., the *imo - v9.8.00000001045* app has an *NIV* of degree 1).
– *NIV* of degree 2: if an app contains at least one path calling the `getIntent()` and the `getParcelableExtra()` APIs (i.e., the percentage of $p\Im_2$ over $p\Im_1$ is greater than zero), it is classified to have an *NIV* of degree 2 (e.g., *imo - v9.8.000000008951* has an NIV of degree 2).
– *NIV* of degree 3: if an app contains at least one path calling the `getIntent()`, the `getParcelableExtra()` and the `startActivity()` APIs (i.e., the percentage of $p\Im_3$ over $p\Im_2$ is greater than zero), it is classified to have an *NIV* of degree 3 (e.g., the *Gmail - v8.5.6.19* app has an NIV of degree 3).
– *NIV* of degree 4: if an app contains at least one path calling the `getIntent()`, the `getParcelableExtra()` and the `startActivity()` APIs and the data flow in this path satisfies the NIV conditions illustrated in Sect. 2 (i.e., the percentage of $\mathcal{NIV}_n$ over $p\Im_3$ is greater than zero), the app has an *NIV* of degree 4.

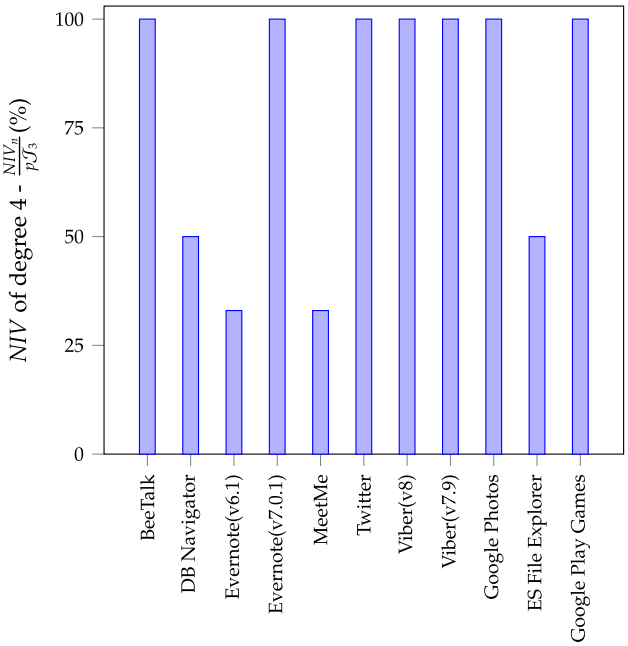

Fig. 5 Apps found to have an *NIV* of degree 4 over the set of 100 apps

In Figs. 3 and 4, we show the apps we found to have an NIV of degree 2 and of degree 3, respectively.

Among the four NIV degrees we defined, the degree 4 is the most significant one, since it illustrates the inner complexity of NIV, but it also motivates the requirements that an NIV detection system should have. Even if an app invokes the three NIV-related APIs in the expected sequence, the app might still be not vulnerable. Thus, an app could invoke the `getIntent()`, `getParcelableExtra()` and `startActivity()` methods, but the `Intent` used to invoke the private `Activity` could be a new one and not the one extracted through the `getParcelableExtra()` method. Figure 5 shows the percentage of execution paths leading to NIV (i.e., NIV_n) over the number of execution paths calling the three NIV-related APIs (i.e., $p\mathcal{J}_3$). As shown in the figure, *DB Navigator*, *Evernote(v6.1)*, *MeetMe* and *ES File Explorer* do not reach the 100% value, which means that they have some paths calling the three NIV-related APIs without actually leading to NIV.

Finally, we also analyzed how the *NIV* evolves over different versions of the same app. Table 6 shows the *NIV* evolution in *Twitter*, *Evernote* and *Viber Messenger*. In all versions, the apps have NIV of degree 1 due to the common use of intent extraction API. While *Twitter* and *Evernote*, first affected by NIV in their initial versions, have been fixed in their latest releases, *Viber Messenger* was and still is affected by NIV. Interestingly, the disappearance of NIV of degree 3 in the latest version of *Evernote (v8)* comes with a lower percentage value for NIV of degree 2.

7 $\mathcal{N}$IV$\mathcal{D}$ evaluation

The purpose of this section is to evaluate the performance of $\mathcal{N}$IV$\mathcal{D}$ by comparing it with the state-of-the-art. In particular, we chose to compare $\mathcal{N}$IV$\mathcal{D}$ with *NIVAnalyzer* [5], which is the most recent solution addressing NIV. To compare $\mathcal{N}$IV$\mathcal{D}$ and *NIVAnalyzer*, we considered the efficacy (Sect. 7.1) and the efficiency (Sect. 7.2). Since the source code of *NIVAnalyzer* is not available in open-source, we implemented it by following the algorithm described in the paper [5].

7.1 Efficacy analysis

The purpose of $\mathcal{N}$IV$\mathcal{D}$ is to determine whether an app is affected by *NIV*. Thus, after running $\mathcal{N}$IV$\mathcal{D}$ against the set of 100 popular apps, we manually double-checked the correctness of our results, to find a 100% precision in the detection. Considering NIV and the design of $\mathcal{N}$IV$\mathcal{D}$, our detection system relies on a deterministic approach and not on a probabilistic one. This means that, once the correctness of the program is proved, there are no false negatives in the final classification. Moreover, over a set of 20,000 apps *NIVAnalyzer* found 203 vulnerable apps according to the NIV discovery module, and 138 vulnerable apps according to the exploitation module. On the contrary, $\mathcal{N}$IV$\mathcal{D}$ has been able to detect nine vulnerable apps out of the 100 analyzed ones.

Table 6 NIV evolution in Twitter, Evernote, and Viber Messenger over different versions

App_version	NIV of degree 1	NIV of degree 2	NIV of degree 3	NIV of degree 4
Twitter (v7.23.0)	Yes	Yes (95.9%)	Yes (0.044%)	Yes
Twitter (v7.48.0)	Yes	Yes (96.9%)	No	No
Evernote (v6.1)	Yes	Yes (43.1%)	Yes (2.6%)	Yes
Evernote (v7.0.1)	Yes	Yes (58.4%)	Yes (0.093%)	Yes
Evernote (v8)	Yes	Yes (35.7%)	No	No
Viber Messenger (v7.9)	Yes	Yes (29.5%)	Yes (1.2%)	Yes
Viber Messenger (v8.9)	Yes	Yes (28.8%)	Yes (1.2%)	Yes

7.2 Efficiency analysis

To compare $\mathcal{NIVD}$ and *NIVAnalyzer* in terms of efficiency, we considered both the number of execution paths undertaken during the analysis and the time required to analyze an app.

NIVAnalyzer is designed to search first for a call to one of the NIV-related APIs, go forward to evaluate the execution paths starting at that call, and eventually move backwards to evaluate the execution paths reaching the same call. Thus, *NIVAnalyzer* identifies and keeps track of both backward and forward paths. For this reason, Figs. 6 and 7 show the comparison of the number of paths analyzed by $\mathcal{NIVD}$ with the backward and forward paths considered by *NIVAnalyzer*, respectively. For 95 apps, the number of execution paths evaluated by $\mathcal{NIVD}$ is less than the number of backward and forward paths analyzed by *NIVAnalyzer*. For two apps both solutions do not analyze any path, for other two apps $\mathcal{NIVD}$ analyzes fewer paths than the forward ones, but more paths than the backward ones, and for 1 app $\mathcal{NIVD}$ analyzes more paths in comparison to both backward and forward ones. The reason for the improvement provided by $\mathcal{NIVD}$ is due to the limited analysis of only the execution paths having calls to the three NIV-related APIs. On the contrary, *NIVAnalyzer* considers all backward and forward paths that start with just one of the three NIV-related APIs. Besides affecting the overall efficiency of the analysis, the approach adopted by *NIVAnalyzer* has also an impact on the accuracy of the classification method. Since *NIVAnalyzer* relies on a threshold value to determine when the analysis needs to be stopped, an evaluation of a high number of not significant execution paths inevitably leads to a lot of false negatives. One significant example of false negative is the *Google Photos* app, not detected by *NIVAnalyzer*.

Considering the second efficiency criterion (i.e., the amount of time), Fig. 8 shows only the apps for which there is a significant difference (i.e., higher than 20%) in the analysis time by $\mathcal{NIVD}$ and by *NIVAnalyzer*. Out of the 100 apps, $\mathcal{NIVD}$ is significantly faster than *NIVAnalyzer* for 34 apps and noticeably slower for 5 others. For the remaining

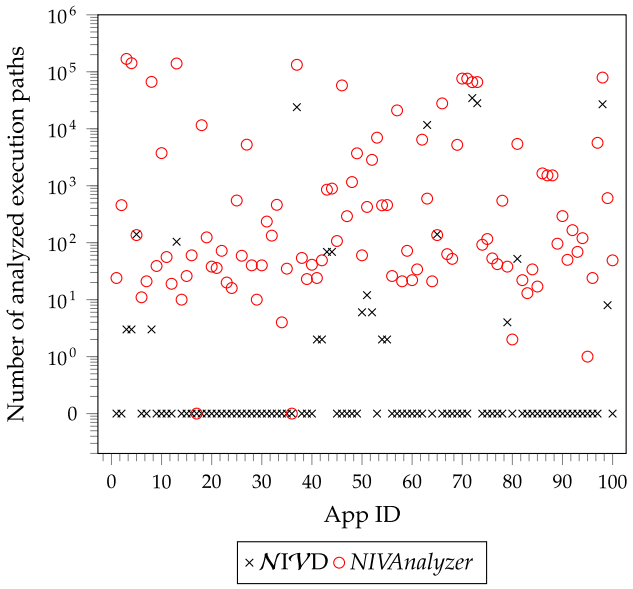

Fig. 6 Comparison between number of execution paths analyzed by $\mathcal{NIVD}$ and number of backward paths analyzed by *NIVAnalyzer*

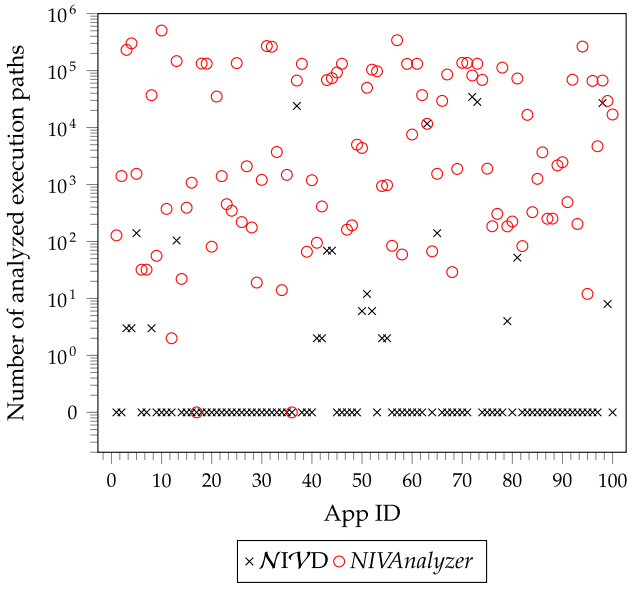

Fig. 7 Comparison between number of execution paths analyzed by $\mathcal{NIVD}$ and number of forward paths analyzed by *NIVAnalyzer*

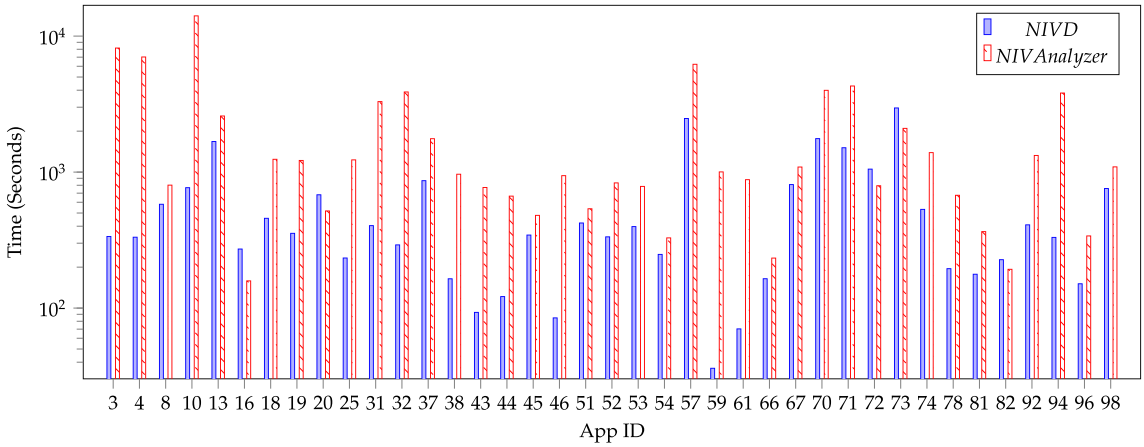

Fig. 8 Apps for which $\mathcal{N}$IV$\mathcal{D}$ is significantly faster or slower than *NIVAnalyzer* in terms of time required to complete the analysis

61 apps, the difference in running times of the two methods is less than 20%. It is worth noting that $\mathcal{N}$IV$\mathcal{D}$ outperforms *NIVAnalyzer* mostly for large size apps, while for smaller ones the approaches have almost similar analysis times.

8 Related work

Analyzing Android apps [13–15] and revealing their vulnerabilities [16–18] have been the main focus of several previous works.

The Amandroid framework [13] relies on a flow and context sensitive static pointer analysis to verify the security of Android apps. According to the authors, with a reasonable amount of additional code (around 100 lines), Amandroid can be used for specialized security analyses. Concerning the application of Amandroid to detect NIV, the authors did not specify how they modified Amandroid to this aim, thus, preventing the comparison with $\mathcal{N}$IV$\mathcal{D}$. Meng et al. [15] presented a tool, called *FindSecurityBugs* and integrated it into the Android Studio IDE, to detect ICC security vulnerability. Unlike $\mathcal{N}$IV$\mathcal{D}$, *FindSecurityBugs* requires the source code of the apps. Based on taint analysis, Xiong et al. [14] presented *IntentSoot*, aimed at revealing `Intent` injection vulnerability in Android apps. *IntentSoot* generates the call graph and the control flow graph of the app under analysis. Then, it analyzes the taint propagation to detect an `Intent` injection vulnerability. The approach is inline with the type system used by $\mathcal{N}$IV$\mathcal{D}$, but it is not able to detect NIV. Component hijacking [19–21] is another Android vulnerability, through which attackers can gain access to private information and damage data integrity. Zhang et al. [20] proposed a method to patch this vulnerability in Android apps, thus, it cannot be compared to $\mathcal{N}$IV$\mathcal{D}$, which purpose is NIV detection.

Wang et al. [4] gave the first proof of feasibility for an NIV-based attack and found that both *Dropbox* and *Facebook* are affected by NIV. However, they did not provide an automatic method for NIV detection, which is instead one of the main purposes of $\mathcal{N}$IV$\mathcal{D}$. Tang et al. [5] designed a static intent flow analysis system (i.e., *NIVAnalyzer*) to automatically detect NIV. *NIVAnalyzer* starts by searching for key points (i.e., APIs such as *getParcelableExtra()*) in the code and it moves the analysis forward only if one of the key points has been found. Then, *NIVAnalyzer* analyzes forward and backward paths that start at key points. The drawback of *NIVAnalyzer* regards both the number of paths to be analyzed and the threshold chosen to stop the analysis. Considering the first, the common use of the NIV-related APIs increases the number of not vulnerable paths under analysis (i.e., paths not including a call to an NIV-related API). Concerning the second limitation, by applying a threshold over the number of paths to be analyzed before interrupting the analysis could skip some vulnerable paths, thus, misleading the overall precision. $\mathcal{N}$IV$\mathcal{D}$ overcomes these drawbacks by analyzing only the required execution paths. Many researchers studied vulnerabilities caused by `Intent` in Android app [22–25]. Wu et al. [23] proposed a technique for analyzing permission leakage vulnerabilities related to ICC. Similarly to $\mathcal{N}$IV$\mathcal{D}$, the approach presented in [23] relies on control flow and data flow analysis to detect suspicious vulnerable paths. However, $\mathcal{N}$IV$\mathcal{D}$ addresses a vulnerability which is not related to permissions and it also handles unsafe control flows generated by Android messaging models. Salva and Zafimiharisoa [25] presented *APSET*, a tool for testing security of apps and able to detect intent-related vulnerabilities. This method uses vulnerability patterns model to represent intent-based vulnerabilities. Such representations are passed as inputs to *APSET* together with the app under testing. However, the formalization of vulnerabilities is not straightforward, which makes *APSET* not easily applicable like $\mathcal{N}$IV$\mathcal{D}$.

9 Conclusion and future work

NIV is a well-known vulnerability, which has been first exploited to build up an attack by Wang et al. [4]. Despite the solutions proposed by previous works to detect the vulnerability, NIV is still infecting Android apps, even the most popular ones. The attraction of NIV for attackers is motivated by the damages it can cause: NIV breaks the Android app model according to which app components are private and not accessible outside the app itself, since it allows any app on a smartphone to access to private components of other apps.

The purpose of this paper is to propose $\mathcal{N}$I$\mathcal{V}$D, a novel method for detecting NIV in Android apps by applying the inference rules of a type system to the smali code of an app. Evaluating $\mathcal{N}$I$\mathcal{V}$D on a set of 100 apps, we found that nine are vulnerable, among which there is also the *Google Photos* app (its detection has been reported to Google under the issue number 124342801). With respect to the state-of-the-art, $\mathcal{N}$I$\mathcal{V}$D is more efficient in terms of time required to analyze an app and number of execution paths considered during the analysis. Concerning the time analysis, $\mathcal{N}$I$\mathcal{V}$D performs significantly faster (20% of improvement) for 34 apps out of the 100 analyzed ones and slightly slower for five apps. Considering the number of analyzed paths, $\mathcal{N}$I$\mathcal{V}$D outperforms *NIVAnalyzer* for 95 apps, has a worse performance for three apps and the same performance for two apps.

Possible future works include to evaluate the existing research solutions (e.g., TaintDroid [8], FlowDroid [7]) as *NIV* detection mechanisms and to measure the *NIV* effects. This direction involves specifying the direct relationship between NIV and other types of vulnerabilities, such as component hijacking. Such relationship enables applying solutions found for one vulnerability to the other one.

Funding This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the LOCARD project (Grant Agreement No. 832735).

Compliance with ethical standards

Conflict of interest All authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

A Sample applications

Table 7 lists the 100 apps downloaded from the Google Play Store and used to evaluate $\mathcal{N}$I$\mathcal{V}$D.

B Result details

Table 8 presents the meaning of each column of Tables 9 and 10 which present the complete analysis undertaken by $\mathcal{N}$I$\mathcal{V}$D over the set of 100 apps.

Table 7 List of apps used for $\mathcal{NIVD}$ evaluation

ID	App (version)	ID	App (version)	ID	App (version)
1	Azar (v3.29.2)	2	BiP Mes (v3.33.13)	3	Gmail (v8.5.6.19)
4	Gmail (v7.11.5)	5	Hangouts (v25.0.19)	6	imo (v9.8.000000010451)
7	imo (v9.8.000000008951)	8	KakaoTalk (v7.2.2)	9	Kik (v13.2.0.8)
10	LINE (v8.6.2)	11	Primo (v1.0.43)	12	Psiphon Pro (v190)
13	QQ (v6.6.7)	14	Google Camera (v4.1.006)	15	4Fun (v1.60)
16	Hello Yo (v1.5.11)	17	IG Hoot (v1.0)	18	Instagram (v48.0)
19	Instagram (v24.0)	20	LinkedIn (v4.1.180)	21	LivU (v1.1.11)
22	musical.ly (v7.3.0)	23	NumberBook (v2.1.4)	24	OmeTV (v6.3.6)
25	Pinterest (v6.68.0)	26	SKOUT (v5.6.2)	27	Snapchat (v10.33.5)
28	Tamago (v1.10.28)	29	Telegram X (v0.20.7)	30	Textfree (v8.17)
31	TextNow (v5.57.0)	32	textPlus (v7.2.3)	33	Tumblr (v10.8.0)
34	veQR (v2.2.0)	35	Weibo (v2.8.1)	36	Assistant (v0)
37	Sheets (v1)	38	OneDrive (v5.10.1)	39	SwiftKey Keyboard (v7)
40	WPS Office (v10)	41	BeeTalk (v3)	42	DB Navigator (v18)
43	Evernote (v6.1)	44	Evernote (v7.0.1)	45	Evernote (v8)
46	Hi There (v11.7)	47	IP Webcam (v1.11.1)	48	IP Webcam (v1.13.25)
49	K-9 Mail (v5.403)	50	MeetMe (v12.10)	51	Polaris Office (v7.3)
52	Twitter (v7.23.0)	53	Twitter (v7.48.0)	54	Viber (v7.9)
55	Viber (v8.9)	56	Candy Crush (v1.113)	57	Clean Master (v6.0.9)
58	IP Webcam(1.5.0)	59	Facebook Lite (v68.0)	60	Firefox (v57.0)
61	Go. Chrome (v62.0.3)	62	Go. Drive (v2.7.372)	63	Go. Photos (v3.9)
64	Go. Play (v8.5.6)	65	Hangouts (v23.0.1)	66	Google Home (v2.8.15.6)
67	Maps Nav. (v9.64.1)	68	Messenger Lite (v19)	69	Snapchat (v10.2)
70	WeChat (v6.6.6)	71	WeChat (v6.5.16)	72	WhatsApp (v2.18.156)
73	WhatsApp (v2.17.395)	74	YouTube (v12.43.52)	75	Air Camera (v1.8.5.1007)
76	Prime Video (v3.0.242.14741)	77	Banggood Shopping (v5.19.2)	78	Beat Ride (v10.37)
79	ES File Explorer (v4.1.9.9.3)	80	GFX Tool (v5.2.2)	81	Google Play Games (v5.14.7825)
82	Google Play Music (v8.18.7847-1.L)	83	Google Translate (v5.26.0.RC02.23-)	84	KineMaster (v4.8.13.12545.GP)
85	MX Player (v1.10.44)	86	Netflix (v6.24.0 build 12 31651)	87	Notepad (v2.0.452)
88	Notes with Caller ID (v1.0.3330)	89	SHAREit (v4.7.8_ww)	90	Spotify (v8.4.94.817)
91	Trip.com (v6.10.3)	92	Truecaller (v10.18.6)	93	UC Browser (v12.8.5.1121)
94	UNICORN (v1.9.2.1)	95	ZArchiver (v0.9.1m)	96	VidMate (v3.6507)
97	TikTok (v5.1.3)	98	Google Docs (v1.19.072.01.30)	99	Google Pay (v2.82.231680166)
100	Google Duo (v47.1.234325686.DR47_RC14)				

Table 8 Notation used for the statistics collected during the $\mathcal{N}$I$\mathcal{V}$D evaluation

Notation	Semantics
$\mathcal{C}_n$	Number of classes
$\mathcal{A}_n$	Number of public Activities
$\mathcal{C}a_n$	Number of classes of public Activities
$\Im_1$	Number of calls to `getIntent()` API in public Activities
$\Im_2$	Number of calls to `getParcelableExtra()` API
$\Im_3$	Number of calls to `startActivity()` and `startActivityForResult()` APIs
$p\Im_1$	Number of paths starting at `getIntent()` (counted in $\Im_1$)
$p\Im_2$	Number of paths that have a sequence of calls to `getIntent()` (counted in $\Im_1$) and `getParcelableExtra()` APIs.
$p\Im_3$	Number of paths that have a sequence of calls to `getIntent()` (counted in $\Im_1$), `getParcelableExtra()`, and `startActivity()` APIs
$\mathcal{N}IV_n$	Number of paths affected by *NIV*
$\mathcal{P}_f$	Number of forward execution paths that start at the invocation of the `getParcelableExtra()` API
$\mathcal{P}_{fs}$	Number of paths in $\mathcal{P}_f$ that include an invocation to `startActivity()`
$\mathcal{P}_b$	Number of backward execution paths that start at the invocation of the `getParcelableExtra()` API
$\mathcal{P}_{bg}$	Number of paths in $\mathcal{P}_b$ that include an invocation to the `getIntent()` API

Table 9 Results obtained after running $\mathcal{N}$I$\mathcal{V}$D over the apps 1–50

ID	$\mathcal{C}_n$	$\mathcal{A}_n$	$\mathcal{C}a_n$	$\Im_1$	$\Im_2$	$\Im_3$	$p\Im_1$	$p\Im_2$	$p\Im_3$	$\mathcal{N}IV_n$	$\mathcal{P}_f$	$\mathcal{P}_{fs}$	$\mathcal{P}_b$	$\mathcal{P}_{bg}$
1	19,263	4	23	12	26	196	69	0	0	0	128	0	24	6
2	23,244	31	197	73	37	585	72,039	0	0	0	1408	0	456	404
3	21,725	22	27	32	91	156	328,292	180	3	0	320,741	29	168,204	133,056
4	19,085	19	23	27	85	140	262,567	84	3	0	299,991	30	141,368	131,650
5	14,167	13	13	22	47	184	89,038	182	140	0	1540	222	136	92
6	9103	9	97	19	9	194	10,498	0	0	0	32	0	11	1
7	7343	8	90	18	8	155	18,575	120	0	0	32	1	21	16
8	28,018	47	460	78	101	1712	133,902	270	3	0	36,922	24	66,491	910
9	20,935	8	32	18	15	116	27,970	72	0	0	56	0	39	27
10	47,521	10	24	27	115	1422	66,394	12	0	0	504,450	7779	3736	3595
11	12,975	3	35	8	48	92	37	8	0	0	373	1	56	19
12	4330	1	21	1	4	67	6	0	0	0	2	0	19	0
13	35,236	27	66	208	165	2419	271,242	3342	104	0	146,514	1066	139,806	117,424
14	7384	3	53	4	8	52	1111	0	0	0	22	2	10	3
15	9715	5	7	10	26	178	64	0	0	0	391	0	26	2
16	14,323	7	210	29	30	293	417,340	0	0	0	1075	1	60	10
17	418	1	1	0	0	10	0	0	0	0	0	0	0	0
18	22,545	7	11	15	44	34	262,185	22	0	0	132,739	0	11,538	10,421
19	16,997	9	13	23	28	126	197,122	1	0	0	132,216	0	124	5
20	28,613	5	25	11	33	432	25	0	0	0	81	2	38	6
21	9927	5	43	10	36	142	152	0	0	0	34998	0	36	4
22	28,799	12	133	68	71	617	4585	3	0	0	1404	10	72	28
23	7552	5	92	8	21	191	739	0	0	0	451	1	20	3
24	4433	2	115	0	14	46	0	0	0	0	346	0	16	2
25	19,819	5	33	18	42	134	11,147	0	0	0	135,155	1	550	488

Table 9 continued

ID	$\mathcal{C}_n$	$\mathcal{A}_n$	$\mathcal{C}a_n$	$\Im_1$	$\Im_2$	$\Im_3$	$p\Im_1$	$p\Im_2$	$p\Im_3$	$\mathcal{NIV}_n$	$\mathcal{P}_f$	$\mathcal{P}_{fs}$	$\mathcal{P}_b$	$\mathcal{P}_{bg}$
26	27,202	21	377	46	47	478	172	0	0	0	219	0	59	14
27	54,192	4	26	13	32	151	153,135	1085	0	0	2086	0	5258	5196
28	16,011	5	34	20	35	189	78	0	0	0	176	2	40	7
29	5658	1	8	1	5	14	22	0	0	0	19	0	10	0
30	19,602	5	29	33	30	376	3918	0	0	0	1206	1	40	5
31	22,326	32	222	26	108	795	103493	0	0	0	269056	1	235	67
32	23,111	6	25	47	44	488	346	0	0	0	262,349	0	133	53
33	22,277	8	26	21	65	248	624	3	0	0	3722	3	462	62
34	3046	64	783	230	4	166	268	3	0	0	14	0	4	1
35	15,377	16	331	41	33	554	1426	15	0	0	1478	8	35	11
36	789	1	1	0	0	7	0	0	0	0	0	0	0	0
37	30,464	22	56	56	53	269	140,909	47,848	23,913	0	66,774	217	132,617	131,338
38	14,263	15	62	34	35	357	63	13	0	0	131,167	574	54	39
39	10,321	3	3	2	16	146	7	0	0	0	66	1	23	9
40	47,372	49	251	91	33	779	8502	3	0	0	1190	11	41	14
41	17,061	20	30	33	25	490	938	2	2	2	95	1	24	5
42	9423	24	40	77	20	167	1252	6	2	1	412	7	49	7
43	6932	24	48	61	21	286	6097	2628	69	23	68,188	51	852	825
44	7121	24	52	70	23	347	125,945	73,627	69	23	73,593	52	891	863
45	14,746	35	57	62	49	514	366,554	131,104	0	0	92,485	3376	107	66
46	7205	3	130	19	28	364	566	3	0	0	131,715	28,090	57,635	57,618
47	2547	3	11	3	5	38	36,873	36,864	0	0	161	0	292	288
48	4054	3	181	5	9	59	131,089	131,072	0	0	192	0	1160	1152
49	3819	7	154	11	30	75	131,324	98,208	0	0	5002	0	3706	3587
50	26,792	22	284	73	50	695	11,482	3582	6	6	4386	6	60	19

Table 10 Results obtained after running $\mathcal{NIVD}$ over the apps 51–100

ID	$\mathcal{C}_n$	$\mathcal{A}_n$	$\mathcal{C}a_n$	$\Im_1$	$\Im_2$	$\Im_3$	$p\Im_1$	$p\Im_2$	$p\Im_3$	$\mathcal{NIV}_n$	$\mathcal{P}_f$	$\mathcal{P}_{fs}$	$\mathcal{P}_b$	$\mathcal{P}_{bg}$
51	29,823	4	44	17	56	546	2443	2110	12	0	49892	13	424	13
52	30,090	11	121	66	117	527	14,206	13,633	6	2	103,718	14	2846	2771
53	33,107	10	77	60	114	535	11,016	10,680	0	0	97,094	15	6971	6898
54	20,610	39	148	69	93	445	535	158	2	2	940	11	453	337
55	23,117	40	156	73	101	474	548	158	2	2	975	12	458	350
56	5490	2	20	1	26	63	9	0	0	0	84	0	26	6
57	30,456	35	608	310	71	748	3,357,127	590,027	0	0	340,934	26	21,009	20,900
58	7347	7	31	3	20	66	8	0	0	0	59	0	21	3
59	3170	2	2	5	10	18	196,897	0	0	0	131,097	0	72	0
60	7769	20	106	16	17	114	2863	0	0	0	7537	3	22	1
61	7843	43	137	39	22	163	15,167	0	0	0	131,364	3	34	1
62	14,507	26	62	89	79	220	386,970	6251	0	0	37,071	534	6431	6361
63	27,745	17	19	49	105	252	47,602	34,995	11,662	11,662	11,627	244	594	508
64	14,584	22	112	38	17	162	66,042	11	0	0	67	1	21	6

Table 10 continued

ID	$\mathcal{C}_n$	$\mathcal{A}_n$	$\mathcal{C}a_n$	$\Im_1$	$\Im_2$	$\Im_3$	$p\Im_1$	$p\Im_2$	$p\Im_3$	$\mathcal{NIV}_n$	$\mathcal{P}_f$	$\mathcal{P}_{fs}$	$\mathcal{P}_b$	$\mathcal{P}_{bg}$
65	13,132	13	13	20	47	188	89,035	182	140	0	1541	223	136	92
66	17,332	6	6	11	58	263	19	0	0	0	29,374	5	27,859	27,828
67	47,469	9	11	2	47	181	6	0	0	0	85,377	8	63	19
68	4748	9	10	19	12	24	152	2	0	0	29	0	52	40
69	40,336	4	32	12	27	131	69,531	0	0	0	1877	0	5216	5190
70	42,260	20	123	130	135	1463	500,462	11,040	0	0	136,484	8	76,030	387
71	35,830	20	115	124	112	1373	301,151	11,040	0	0	136,126	6	75,880	251
72	9970	26	283	67	28	507	903,010	186,371	34,561	0	81,886	13,826	65,782	65,717
73	8675	24	214	78	18	427	1,274,592	168,626	28,104	0	131,668	9386	66,033	65,964
74	26,718	12	13	15	27	120	28,549	2286	0	0	68,576	86	92	9
75	17,583	18	79	72	41	480	197,728	26	0	0	1897	0	116	13
76	23,630	25	404	42	25	200	65,969	0	0	0	185	3	53	19
77	12,741	9	140	23	37	202	144	0	0	0	305	0	42	10
78	14,362	1	12	1	41	161	1	0	0	0	112,603	0	546	490
79	19,231	27	316	81	32	395	197,039	118	4	2	184	12	38	15
80	4725	1	3	0	2	20	0	0	0	0	224	0	2	0
81	12,808	46	59	53	51	90	55,320	23,052	52	52	72,660	54	5415	5348
82	15,923	22	114	38	18	166	66,067	11	0	0	83	1	22	7
83	8432	8	9	11	9	96	131,163	0	0	0	16,695	0	13	7
84	22,319	6	35	21	31	237	85	0	0	0	327	0	34	6
85	18,300	22	160	44	20	201	83,336	0	0	0	1256	3	17	2
86	17,989	6	23	3	47	196	6	0	0	0	3669	0	1646	637
87	12,005	1	28	0	8	170	0	0	0	0	251	0	1519	0
88	12,301	1	23	1	10	176	3	0	0	0	252	0	1520	0
89	20,175	20	246	80	40	442	25,529	1	0	0	2165	0	96	16
90	34,615	11	44	18	67	249	92	17	0	0	2461	49	293	39
91	36,257	6	33	6	48	631	70	0	0	0	490	0	50	11
92	25,075	12	42	35	57	428	269,379	0	0	0	69,137	11,534	166	110
93	23,062	14	336	13	27	241	34,860	0	0	0	203	0	69	2
94	22,339	3	9	7	27	226	14	0	0	0	264,040	0	120	75
95	301	3	43	7	1	20	14	0	0	0	12	0	1	0
96	11,341	11	58	14	19	313	42	0	0	0	65,828	0	24	6
97	44,511	11	104	74	87	910	308	16	0	0	4688	22	5663	5514
98	28,269	22	69	69	60	276	138,866	55,065	26,900	0	66,818	72	79,159	79,062
99	17,392	14	130	49	58	389	12,493	360	8	0	29,266	6	606	128
100	11,527	13	15	27	23	85	116,078	88	0	0	16,932	2	49	11

References

1. G DATA Blog: Some 343 new android malware samples every hour in 2017. https://www.gdatasoftware.com/blog/2018/02/30491-some-343-new-android-malware-samples-every-hour-in-2017. Accessed Mar 2019
2. Özkan, S.: CVE details: the ultimate security vulnerabilities datasource. https://www.cvedetails.com/index.php. Accessed 4 2019
3. Parnika, P., Dutta, K.: A survey on various threats and current state of security in android platform. ACM Comput. Surv. **52**(1), 21 (2019)
4. Wang, R., Xing, L., FengWang, X., Chen, S.: Unauthorized origin crossing on mobile platforms: threats and mitigation. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 635–646. ACM (2013)
5. Tang, J., Cui, X., Zhao, Z., Guo, S., Xu, X., Hu, C., Ban, T., Mao, B.: Nivanalyzer: a tool for automatically detecting and verifying next-intent vulnerabilities in android apps. In: IEEE International Conference on Software Testing, Verification and Validation, pp. 492–499 (2017)
6. Chin, E., Porter Felt, A., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, pp. 239–252. ACM (2011)

7. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Not. **49**(6), 259–269 (2014)
8. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B., Cox, L., Jung, J., McDaniel, P., Sheth, A.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32**(2), 1–29 (2014)
9. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: semantics-based detection of android malware through static analysis. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 576–587 (2014)
10. Android Developers: Developer guides: intents and intent filters. https://developer.android.com/guide/components/intents-filters.html. Accessed in 2018
11. Desnos, A.: Android-androguard: a full python tool to play with android files (2011). https://github.com/androguard/androguard/. Accessed in 2018
12. Configure an Android Device: Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Accessed in 2019
13. Wei, F., Roy, S., Ou, X.: Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Trans. Privacy Secur. (TOPS) **21**(3), 14 (2018)
14. Xiong, B., Xiang, G., Du, T., He, J.S., Ji, S.: Static taint analysis method for intent injection vulnerability in android applications. In: International Symposium on Cyberspace Safety and Security, pp. 16–31 (2017)
15. Meng, X., Qian, K., Lo, D., Bhattachrya, P.: Detectors for intent ICC security vulnerability with android IDE. In: International Conference on Ubiquitous and Future Networks, pp. 355–357 (2018)
16. Joshi, J., Parekh, C.: Android smartphone vulnerabilities: a survey. In: International Conference on Advances in Computing, Communication, and Automation, pp. 1–5 (2016)
17. Davi, L., Dmitrienko, A., Sadeghi, A., Winandy, M.: Privilege escalation attacks on android. In: International Conference on Information Security, pp. 346–360 (2010)
18. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: ACM Conference on Computer and Communications Security, pp. 229–240 (2012)
19. Li, L., Bartel, A., Bissyandé, T., Klein, J., Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: Iccta: detecting inter-component privacy leaks in android apps. In: The 37th International Conference on Software Engineering-Volume 1, pp. 280–291 . IEEE Press (2015)
20. Zhang, M., Yin, H.: Appsealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. IN: Network and Distributed System Security Symposium (2014)
21. Ahmad, M., Costamagna, V., Crispo, B., Bergadano, F.: Teicc: targeted execution of inter-component communications in android. In: ACM Symposium on Applied Computing, pp. 1747–1752 (2017)
22. Gallingani, D., Gjomemo, R., Venkatakrishnan, V., Zanero, S.: Practical exploit generation for intent message vulnerabilities in android. In: ACM Conference on Data and Application Security and Privacy, pp. 155–157 (2015)
23. Wu, S., Zhang, Y., Jin, B., Cao, W.: Practical static analysis of detecting intent-based permission leakage in android application. In: IEEE 17th International Conference on Communication Technology, pp. 1953–1957 (2017)
24. Zhang, J., Yao, Y., Li, X., Xie, J., Wu, G.: An android vulnerability detection system. In: International Conference on Network and System Security, pp. 169–183 (2017)
25. Salva, S., Zafimiharisoa, S.: Apset, an android application security testing tool for detecting intent-based vulnerabilities. Int. J. Softw. Tools Technol. Transf. **17**(2), 201–221 (2015)